Moving to Camunda Cloud

2022-03-24T12:45:00-04:00

## Some background

I’ve written a bit about using [Camunda](https://camunda.com?ref=davidgsiot) to do all sorts of things from [automating IoT processes](https://davidgs.com/posts/category/camunda/automating-iot-camunda/) to helping with the management of a Slack community. Ok, so I haven’t written about all of those things, but I’ve certainly *done* them.

In all of those projects I’ve used [Camunda](https://camunda.com?ref=davidgsiot) Platform 7 to deploy and run my processes, but most of the actual work was done by external tasks which I wrote in Golang. This is decidedly *not* the way I was supposed to do things. Most Camunda Platform users write everything in Java. I’ve been using Java since before it was actually released by Sun Microsystems back in 1995 (a moment of silence please for a fantastic company that changed the industry, please).

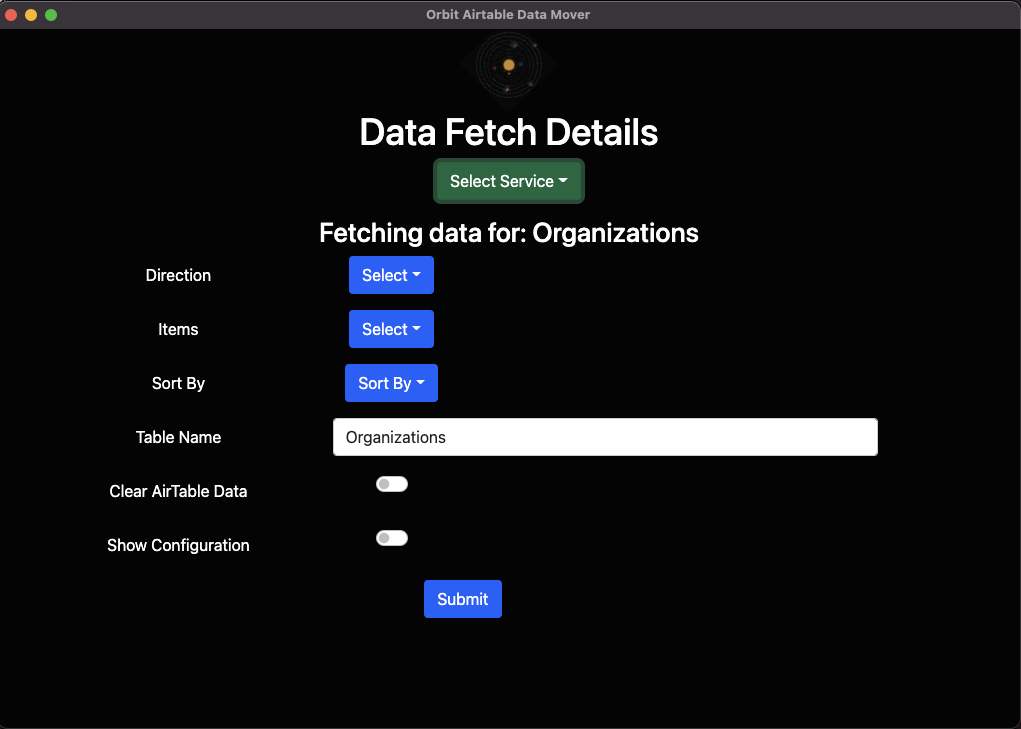
I was a “Java Technologist” back in 1996, a job we’d now call either an evangelist or a Developer Advocate. But that’s not the point. The point is that even with that history with Java I haven’t actually written any meaningful Java code in over 10 years. But I now write lots of Go code, so here’s where we are.

Last bit of background: I’ve been trying to teach myself React.js lately with some (mostly limited) success. So when another team member asked for help automating moving data from [Orbit](https://orbit.love) to [Airtable](https://airtable.com) I thought I’d see if I could write a desktop React App to do it.

## Writing the React App

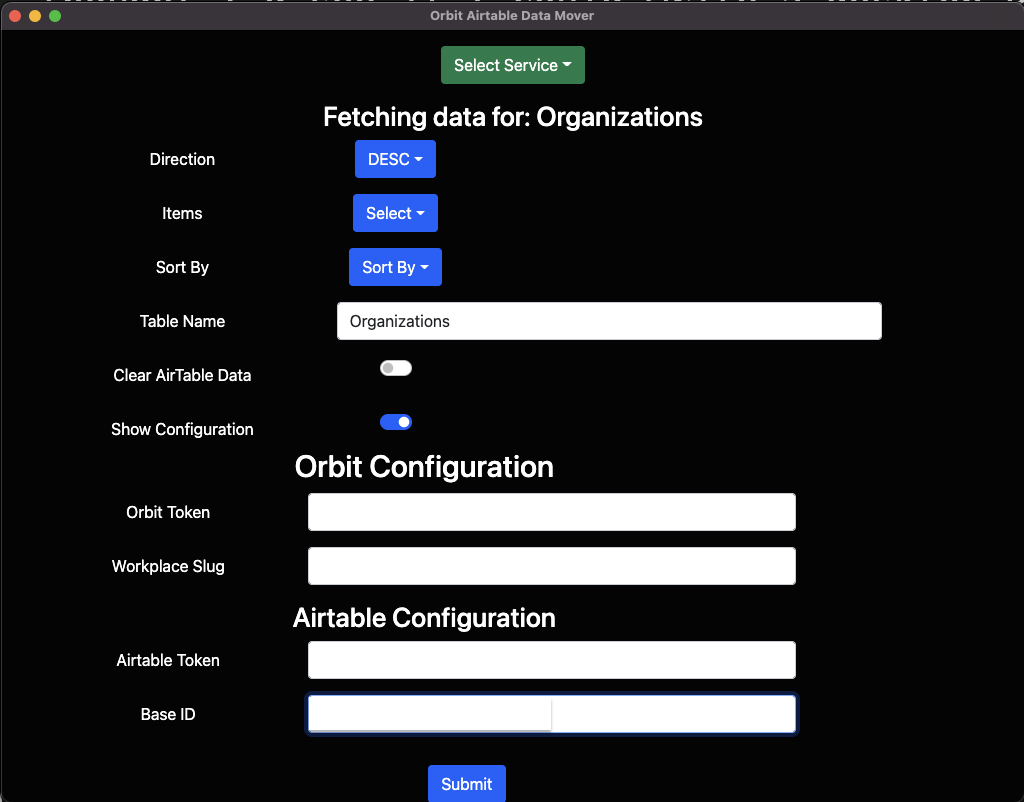
It really wasn’t a complicated application to write. Call the Orbit API to get the data requested, reformat it a bit, and then call the Airtable API to save it. Pretty straightforward.

Here’s what the app looks like:



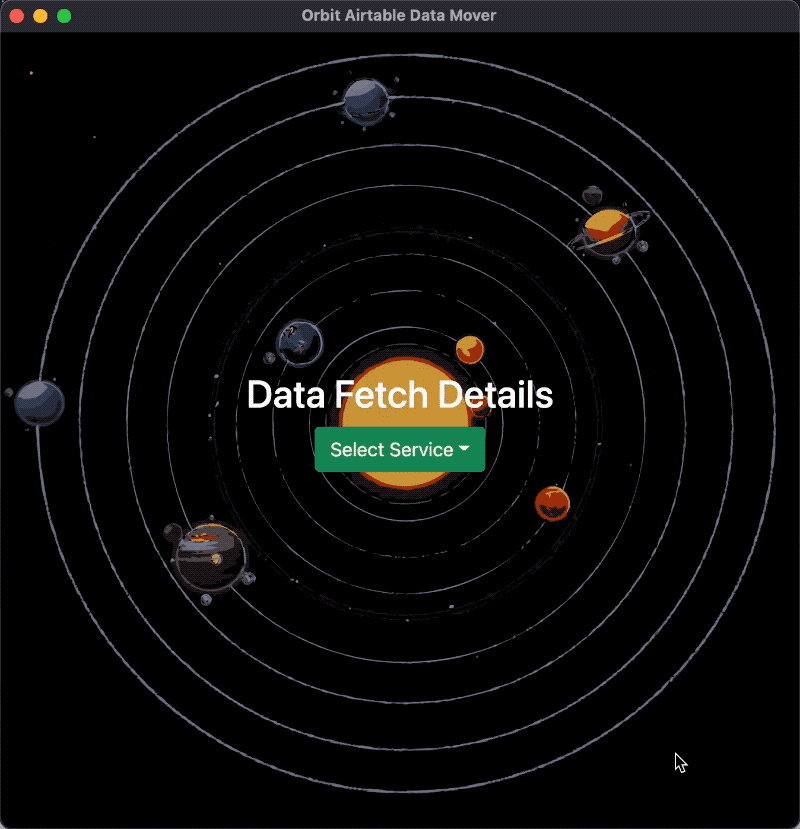
The application’s interface

Nice and simple. And there’s a little slider that will show you the configuration of the app for things like the authentication tokens, etc.



The application’s configuration

As a side note, I chose violence one day and made this the UI:



The UI on a bad day

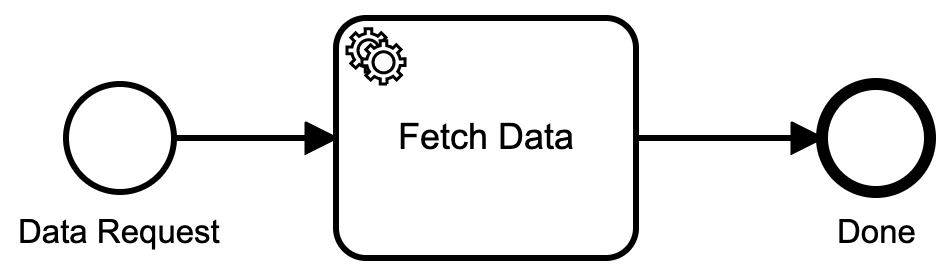
I’m not proud of the choices I made that day.

## Giving Camunda Cloud a whirl

The day after I delivered the app to my co-worker, she came back and said “[Mary](https://twitter.com/mary_grace) asked if this uses Camunda Cloud.”

I had, originally, made the app using Camunda Platform 7, but getting the React App to post directly to Camunda Platform was proving problematic, so I just bypassed Camunda altogether.

But since I was asked *if* I was using Camunda Cloud, I decided to see if I *could* use Camunda Cloud! So first I came up with this super-complicated BPMN diagram:



The Diagram

Doesn’t get more simple than that, does it?

I deployed that simple process to Camunda Cloud, and then set about writing the handlers for it.

## Handling Camunda Cloud

I was super happy to see that one of the libraries available for Camunda Cloud was a Golang library! Oh, happy day!!

import (  
 "github.com/camunda-cloud/zeebe/clients/go/pkg/entities"  
 "github.com/camunda-cloud/zeebe/clients/go/pkg/worker"  
 "github.com/camunda-cloud/zeebe/clients/go/pkg/zbc"  
)

Gave me all the Go goodness I would need for connecting to Camunda Cloud. One thing I encountered was that the Go library assumes that all the variables for connecting to the cloud are saved in environment variables. I neglected to notice this at first, so I saved all my credentials in a config.yaml file and … it still didn’t work. Oh yeah, environment variables.

type ENV struct {  
 ZeebeAddress string `yaml:"zeebeAddress"`  
 ZeebeeClientID string `yaml:"zeebeeClientID"`  
 ZeebeClientSecret string `yaml:"zeebeeClientSecret"`  
 ZeebeAuthServer string `yaml:"zeebeAuthServer"`  
}  
  
var config = ENV{}  
  
func init(){  
 dat, err := ioutil.ReadFile("path/to/config/zeebe.yaml")  
 if err != nil {  
 log.Fatal("No startup file: ", err)  
 }  
 err = yaml.Unmarshal(dat, &config)  
 if err != nil {  
 log.Fatal(err)  
 }  
 config.ZeebeAddress = os.Getenv("ZEEBE\_ADDRESS")  
 if config.ZeebeAddress == "" {  
 a.init\_proc()  
 os.Setenv("ZEEBE\_ADDRESS", config.ZeebeAddress)  
 os.Setenv("ZEEBE\_CLIENT\_ID", config.ZeebeeClientID)  
 os.Setenv("ZEEBE\_CLIENT\_SECRET", config.ZeebeClientSecret)  
 os.Setenv("ZEEBE\_AUTH\_SERVER", config.ZeebeAuthServer)  
 }  
 client, err := zbc.NewClient(&zbc.ClientConfig{  
 GatewayAddress: config.ZeebeAddress,  
 })  
 if err != nil {  
 panic(err)  
 }  
 jobWorker := client.NewJobWorker().JobType("fetch\_data").Handler(a.handleJob).Open()  
 go func() {  
 <- readyClose  
 jobWorker.Close()  
 jobWorker.AwaitClose()  
 }()  
}

I decided to just keep that little configuration bit since I would be running this process as a system service, and I didn’t want to muck about with environment variables for a system service.

Once I had the client initialized, I had to set up a handler for when a process kicked off (I’ll get to how I kicked off the process in a minute). The process handler jobWorker listens for tasks called fetch\_data and when it gets one, it calls handleJob to take care of it. It uses a channel inside a function so that I can handle multiple requests simultaneously, if I need to.

## Starting a process

Since I was having trouble with Camunda Platform and CORS headers, I needed to write a server process that could deal with the incoming requests from the application.

// The URLs I will accept, handle OPTIONS for CORS  
func (a \*App) InitializeRoutes() {  
 a.Router.HandleFunc("/myEndPoint", a.handleOrgs).Methods("OPTIONS", "POST")  
}  
  
// Run it!  
func (a \*App) Run(addr string) {  
 credentials := handlers.AllowCredentials()  
 handlers.AllowedHeaders([]string{"X-Requested-With", "Content-Type", "Authorization", "Referer", "Origin"})  
 methods := handlers.AllowedMethods([]string{"POST", "GET", "OPTIONS"})  
 origins := handlers.AllowedOriginValidator(originValidator)  
 log.Fatal(http.ListenAndServeTLS(addr, cert, key, handlers.CORS(credentials, methods, origins, handlers.IgnoreOptions())(a.Router)))  
}  
  
// handle the CORS preflight request  
func (a \*App) handleCORS(w http.ResponseWriter, r \*http.Request) {  
 w.Header().Set("Access-Control-Allow-Origin", "\*")  
 w.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")  
 w.Header().Set("Access-Control-Allow-Headers", "Accept, Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token, Authorization")  
}  
  
// handle the incoming request  
func (a \*App) handleOrgs(w http.ResponseWriter, r \*http.Request) {  
 if r.Method == "OPTIONS" {  
 a.handleCORS(w, r) // preflight  
 return  
 }  
 if r.Header.Get("Content-Type") != "" {  
 value, \_ := header.ParseValueAndParams(r.Header, "Content-Type")  
 if value != "application/json" {  
 msg := "Content-Type header is not application/json"  
 http.Error(w, msg, http.StatusUnsupportedMediaType)  
 return  
 }  
 }  
 body, err := ioutil.ReadAll(r.Body)  
 if err != nil {  
 fmt.Println(err)  
 }  
 // limit to 1MB  
 r.Body = http.MaxBytesReader(w, r.Body, 1048576)  
 pdat := ProcessData{}  
 err = json.Unmarshal(body, &pdat)  
 dec := json.NewDecoder(r.Body)  
 if err != nil {  
 var syntaxError \*json.SyntaxError  
 var unmarshalTypeError \*json.UnmarshalTypeError  
 switch {  
 // Catch any syntax errors in the JSON  
 case errors.As(err, &syntaxError):  
 msg := fmt.Sprintf("Request body contains badly-formed JSON (at position %d)", syntaxError.Offset)  
 http.Error(w, msg, http.StatusBadRequest)  
 // In some circumstances Decode() may also return an  
 // io.ErrUnexpectedEOF error for syntax errors in the JSON.  
 case errors.Is(err, io.ErrUnexpectedEOF):  
 msg := "Request body contains badly-formed JSON"  
 http.Error(w, msg, http.StatusBadRequest)  
 // Catch any type errors We can interpolate the relevant  
 // field name and position into the error  
 // message to make it easier for the client to fix.  
 case errors.As(err, &unmarshalTypeError):  
 msg := fmt.Sprintf("Request body contains an invalid value for the %q field (at position %d)", unmarshalTypeError.Field, unmarshalTypeError.Offset)  
 http.Error(w, msg, http.StatusBadRequest)  
 // Catch the error caused by extra unexpected fields in the request body  
 case strings.HasPrefix(err.Error(), "json: unknown field "):  
 fieldName := strings.TrimPrefix(err.Error(), "json: unknown field ")  
 msg := fmt.Sprintf("Request body contains unknown field %s", fieldName)  
 http.Error(w, msg, http.StatusBadRequest)  
 // An io.EOF error is returned by Decode() if the request body is  
 // empty.  
 case errors.Is(err, io.EOF):  
 msg := "Request body must not be empty"  
 http.Error(w, msg, http.StatusBadRequest)  
 // Catch the error caused by the request body being too large.  
 case err.Error() == "http: request body too large":  
 msg := "Request body must not be larger than 1MB"  
 http.Error(w, msg, http.StatusRequestEntityTooLarge)  
 // Otherwise default to logging the error and sending a 500 Internal  
 // Server Error response.  
 default:  
 log.Println(err.Error())  
 http.Error(w, http.StatusText(http.StatusInternalServerError), http.StatusInternalServerError)  
 }  
 return  
 }  
 // Call decode again, using a pointer to an empty anonymous struct as  
 // the destination. If the request body only contained a single JSON  
 // object this will return an io.EOF error. So if we get anything else,  
 // we know that there is additional data in the request body.  
 err = dec.Decode(&struct{}{})  
 if err != io.EOF {  
 msg := "Request body must only contain a single JSON object"  
 http.Error(w, msg, http.StatusBadRequest)  
 return  
 }  
 // error free, we can start the process  
 err = startProcess(pdat)  
 if err != nil {  
 fmt.Println(err)  
 http.Error(w, err.Error(), http.StatusInternalServerError)  
 return  
 }  
}  
  
func startProcess(pdat ProcessData) error {  
 client, err := zbc.NewClient(&zbc.ClientConfig{  
 GatewayAddress: config.ZeebeAddress,  
 })  
 if err != nil {  
 return err  
 }  
 // turn the data structure into a map, which is what the Zeebe API expects  
 var b map[string]interface{}  
 inter, err := json.Marshal(pdat)  
 if err != nil {  
 return err  
 }  
 json.Unmarshal([]byte(inter), &b)  
 // create the process  
 ctx := context.Background()  
 request, err := client.NewCreateInstanceCommand().BPMNProcessId("orbit-data").LatestVersion().VariablesFromMap(b)  
 if err != nil {  
 return err  
 }  
 msg, err := request.Send(ctx)  
 if err != nil {  
 return err  
 }  
 return nil  
}

That’s all it took to accept incoming requests from the Application and then start a process in Camunda Cloud.

## Handling the task completion

Now that I have a way to start the process, I need to handle the tasks as they happen.

If you remember from earlier I had set up a task handler for the process:

jobWorker := client.NewJobWorker().JobType("fetch\_data").Handler(a.handleJob).Open()  
 go func() {  
 <- readyClose  
 jobWorker.Close()  
 jobWorker.AwaitClose()  
 }()

So now it’s time to write that whole handleJob thing.

func (a \*App) handleJob(client worker.JobClient, job entities.Job){  
 jobKey := job.GetKey()  
 \_, err := job.GetCustomHeadersAsMap()  
 if err != nil {  
 a.failJob(client, job)  
 return  
 }  
 // get all the submitted variables  
 variables, err := job.GetVariablesAsMap()  
 if err != nil {  
 a.failJob(client, job)  
 return  
 }  
 request, err := client.NewCompleteJobCommand().JobKey(jobKey).VariablesFromMap(variables)  
 if err != nil {  
 a.failJob(client, job)  
 return  
 }  
 incomingData := ProcessData{}  
 jsonStr, err := json.Marshal(variables)  
 if err != nil {  
 fmt.Println(err)  
 }  
 err = json.Unmarshal(jsonStr, &incomingData)  
 if err != nil {  
 fmt.Println("Json unmarshall: ", err)  
 }  
 // this is where I get the data from Orbit, and send it to Airtable.  
 err = handleProcess(incomingData)  
 if err != nil {  
 a.failJob(client, job)  
 return  
 }  
 // If all of that works, complete the job  
 ctx := context.Background()  
 \_, err = request.Send(ctx)  
 if err != nil {  
 panic(err)  
 }  
 log.Println("Successfully completed job")  
 //close(readyClose)  
}  
  
// Handle failing a job  
func (a \*App) failJob(client worker.JobClient, job entities.Job) {  
 log.Println("Failed to complete job", job.GetKey())  
 ctx := context.Background()  
 \_, err := client.NewFailJobCommand().JobKey(job.GetKey()).Retries(job.Retries - 1).Send(ctx)  
 if err != nil {  
 panic(err)  
 }  
}

That is pretty much it! I won’t bore you with all the shenanigans I had to go through in order to get the data out of Orbit and into Airtable since that’s not entirely relevant to the Camunda Cloud process.

## A complaint against Airtable

I will levy a large complain against the Airtable API for deleting records from a table. Ok, maybe 2.

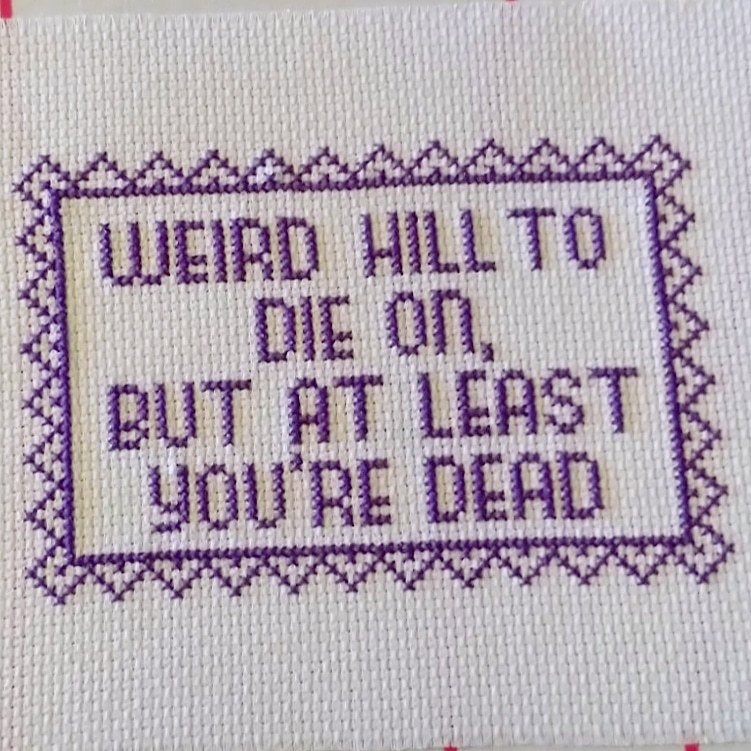
1. There is no way to just clear all the data from a table. You can only delete 10 records at a time, and you have to first fetch all the data from the table in order to get the record IDs first. Then go delete them 10 at a time. This is a waste of time and resources.
2. The Airtable API for deleting records is garbage.

The docs say: > To delete Table records, issue a DELETE request to the Table endpoint. Note that table names and table ids can be used interchangeably. Using table ids means table name changes do not require modifications to your API request. > > Your request should include a URL-encoded array of up to 10 record IDs to delete.

And the sample code supplied by Airtable is:

curl -v -X DELETE https://api.airtable.com/v0/BASE\_ID/TABLE\_NAME \  
 -H "Authorization: Bearer YOUR\_API\_KEY" \  
 -G \  
 --data-urlencode 'records[]=rec9mP3czPxkvf9IR' \  
 --data-urlencode 'records[]=recMxJ0texTTI5BPq'

I’m assuming you can see the problem here. **That is not an array of record IDs!!** You have to put each record ID on a separate line, and then send it all as application/x-www-form-urlencoded data. And for some dumb reason, the URL parameter **must** be called records[]. I guess they decided to add the [] so they could call it an array. It’s still not an array. It just isn’t. This is a hill I’ll die on.



Weird hill to die on, but at least you’re dead

I lost an hour of my life on this.

func deleteNow(delData AirtableData) error {  
 recordCounter := 0  
 records := make([]string, 10)  
 // delData is a struct{} that holds all the records to delete  
 // All of this is because the API doesn't actually take an array  
 for \_, record := range delData.Records {  
 records[recordCounter] = "records[]=" + record.ID  
 recordCounter++  
 if recordCounter == 10 {  
 urlParm := strings.Join(records, "&")  
 err = deleteNow(urlParm, incoming)  
 if err != nil {  
 return err  
 }  
 recordCounter = 0  
 records = make([]string, 10)  
 }  
 }  
 if recordCounter > 0 {  
 urlParm := strings.Join(records, "&")  
 err = deleteNow(urlParm, incoming)  
 if err != nil {  
 return err  
 }  
 }  
 return nil  
}  
  
func deleteNow(urlParm string, incoming AirtableData) error {  
 client := &http.Client{}  
 delReq, err := http.NewRequest("DELETE", "https://api.airtable.com/v0/" + incoming.BaseID + "/" + incoming.TableName + "?" + urlParm, nil)  
 if err != nil {  
 return err  
 }  
 parseFormErr := delReq.ParseForm()  
 if parseFormErr != nil {  
 fmt.Println(parseFormErr)  
 }  
 delReq.Header.Add("Authorization", "Bearer " + incoming.AirtableToken)  
 delReq.Header.Add("Content-Type", "application/x-www-form-urlencoded; charset=utf-8")  
 \_, err = client.Do(delReq)  
 if err != nil {  
 fmt.Println("Failure : ", err)  
 }  
 return nil  
}

So if you are also trying to delete records from an Airtable table, I just solved it for you. Ignore their API docs.

## Conclusions

Camunda Cloud basically runs almost everything as an external task, which can all be written in Golang. Since this is how I was doing everything before anyway, Camunda Cloud is going to be my default from now on! I may even re-write a bunch of my Camunda Platform processes to be Camunda Cloud processes, since all the task handling is already done in Go already.

At least for me, this new way of implementing things is very natural and makes a huge amount of sense. It fits right into how I already work, so it’s a slam-dunk for me to keep doing it!

I’d love to hear what you think about this new way of doing things, so feel free to leave comments, etc.!